A Hybrid Approach for Exact Coloring of Massive Graphs

Emmanuel Hebrard[1] and George Katsirelos[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, France, email: hebrard@laas.fr
[2] MIAT, UR-875, INRA, France, email: gkatsi@gmail.com **

Abstract. The graph coloring problem appears in numerous applications, yet many state-of-the-art methods are hardly applicable to real world, very large, networks. The most efficient approaches for massive graphs rely on "peeling" the graph of its low-degree vertices and focus on the maximum k-core where k is some lower bound on the chromatic number of the graph. However, unless the graphs are extremely sparse, the cores can be very large, and lower and upper bounds are often obtained using greedy heuristics.
In this paper, we introduce a combined approach using local search to find good quality solutions on massive graphs as well as locate small subgraphs with potentially large chromatic number. The subgraphs can be used to compute good lower bounds, which makes it possible to solve optimally extremely large graphs, even when they have large k-cores.

1 Introduction

The *Vertex Coloring Problem* (VCP) asks for the minimum number of colors that can take the vertices of a graph G so that no two adjacent vertices share a color. This number $\chi(G)$ is called the *chromatic number* of the graph.

The VCP has numerous applications. For instance, when allocating frequencies, devices on nearby locations should work on different frequencies to avoid interference. The chromatic number of this distance-induced graph is thus the minimum span of required frequencies [1, 22]. In compilers, finding an optimal register allocation is a coloring problem on an interference graph of value live ranges [6]. In timetabling, assigning time slots to lectures so that no two classes attended by a common subset of student happen in parallel is a VCP [8].

The best performing approaches to the VCP often do not scale to extremely large graphs such as, for instance, social networks. In fact, on networks with several million nodes, even local search methods are seldom used and the best approaches rely on scale reduction and greedy heuristics both for lower and upper bounds [16, 27]. Indeed, the main technique used for reducing the graph consists in removing vertices of degree lower than some lower bound on the chromatic number. This technique might be very effective on sparse graphs especially when

** The second author was partially supported by the french "Agence nationale de la Recherche", project DEMOGRAPH, reference ANR-16-C40-0028.

a maximum or a maximal clique provides a good lower bound. Several real-world extremely large sparse graphs can be efficiently tackled, even via complete algorithms, after such preprocessing. However, even relatively sparse graphs can have a large core of vertices whose degree within the core is higher than the chromatic number. In this case, there are not many practical techniques for upper bounds and most proposed approaches rely on greedy heuristics, in particular Brelaz' DSATUR [5]. Likewise, in this context there is virtually no method for computing a lower bound other than finding a large clique in the graph. As a result, there is little hope to optimally solve an instance with a large core, and whose chromatic number is strictly larger than the size of its largest clique.

In this paper we consider two datasets of very large graphs. The first set, denoted `dimacs10`, contains 30 graphs from the 10th DIMACS implementation challenge [3]. It consists of two subclasses, the former containing graphs with heavy-tailed distribution of degrees and the latter contains quasi-regular graphs. The second set, reffered to as `snap`, contains 75 graphs from the Stanford Large Network Dataset Collection [15]. These graphs correspond to social, citation, collaboration, communication, road or internet networks. They range from tens of thousands to several million vertices and all have extremely low density.

Whereas about half of these graphs are easy or even trivial for the state-of-the-art approaches, the other half remain too large and hard to color even after preprocessing. We show that by combining several methods including local search, heuristics and complete algorithms, we can close a significant proportion (close to 40%) of these hardest instances, even if they contain hundreds of thousands of vertices after preprocessing and even if their chromatic number is larger than their clique number. We survey the related work in section 2, describe our main contribution, a method to obtain good lower bounds on very large graphs in section 3 and an effective local search approach to obtain good upper bounds in section 4. Finally we report the results of an experimental comparison of our method with the state of the art on the datasets described above.

2 Related Work

Heuristic methods are very relevant since they easily scale to very large inputs. In particular, the DSATUR heuristic proposed by Brelaz [5] is instrumental in the state-of-the-art method on the datasets we consider, `FastColor` [16]. The DSATUR heuristic builds a coloring C mapping vertices to colors. It iteratively choses a vertex from a set U initially containing all vertices V of the graph. The chosen vertex v is the one with maximum *saturation degree* $\delta^{sat}(v)$ defined as the number of colors among its neighbors $N(v)$, i.e., $\delta^{sat}(v) = |\{C(u) \mid u \in (N(v) \setminus U)\}|$. In case of a tie, the vertex with maximum degree $|N(v)|$ is selected. Then it sets $C(v)$ to the smallest possible color $\min(\mathbb{N} \setminus \{C(u) \mid u \in (N(v) \setminus U)\})$.

Notice that DSATUR-based branch and bound algorithms [9, 25] are among the best complete methods, alongside column generation approaches [18, 19] and SAT-based models and hybrid algorithms [11, 24, 26, 29]. However, none of these scale to graphs with more than a few thousands vertices.

2.1 Local Search

Local search and meta heuristics have long been applied to graph coloring (e.g. [12]), and with great success. All the best known colorings on the commonly used dataset from the second dimacs challenge [13] were obtained by such methods[3].

In principle, local search approaches seem very well suited for coloring large graphs, and indeed most algorithms scale very well to relatively large graphs. However, surprisingly, we could not find a report of a local search or a meta-heuristic approach applied to the large graphs of the `snap` and `dimacs10` datasets, or on graphs of similar magnitude.

When the number of vertices grows really large, then one must be very careful about the implementation details. As a matter of fact, several off-the-shelf algorithms we tried used data structures with a space complexity quadratic in the number of vertices, and are de facto irrelevant. Another critical point is the size of the neighborhood. For instance, the most common tabu scheme considers all the (non-tabu) moves of any node sharing a color with a neighbor, to a different color. Typical methods evaluate every such move and choose the one that decreases the most the number of conflicts. The number of such moves to consider in a graph with millions of vertices can be prohibitive, especially when starting from low quality initial solutions. The state-of-the-art memetic algorithm `HEAD` [20] uses a similar tabu search, and although we made superficial changes to make it capable of loading massive graphs in memory, it performed poorly on those. After a non-exhaustive review of the literature and of the available software, our belief is that if most of these methods *could* be adapted to extremely large and sparse graphs, however, it would require some non-trivial implementation work.

Blöchliger and Zufferey's local search algorithm [4] appears to be relatively promising in this context. The idea is to try to complete a partial coloring, i.e., a partition of the vertices into of k disjoint independent sets $\{C_1, \ldots, C_k\}$ plus an extra set U of "uncolored" vertices. A move consists in swapping a node $v \in U$ with the vertices $N(v) \cap C_i$ for some color $i \in \{1, \ldots, k\}$. A move (v, i) minimizing $|N(v) \cap C_i|$ is randomly chosen. In order to escape local minima, after each move (v, i), the moves (u, i) for $u \in N(v)$ are added to a tabu list so that v will stay with color i for a given number of iterations. When the set U becomes empty, a k-coloring is obtained and the process can continue by randomly eliminating one color i, that is, setting $U = C_i$ and removing C_i from the partition.

2.2 Independent Set Extraction

Whereas sequence-based coloring heuristics (such as DSATUR) explore the vertices and insert them into the smallest possible color class (or independent set), Leighton's `RLF` heuristic [14] extracts one maximal independent set (or color class) at a time. This technique has been shown to be more effective than DSATUR on some graphs, however it has a higher computational cost.

[3] `http://www.info.univ-angers.fr/~porumbel/graphs/`

Recent effective methods for large graphs rely on this principle. For instance, Hao and Wu [10] recently proposed a method which iteratively extracts maximal independent sets until the graphs contains no more than a given number of vertices. Then, any algorithm can be used on the residual graph to produce a k-coloring which can be trivially extended to a $k+p$-coloring of the whole graph if p independent sets have been extracted. Moreover, the authors show that it may be effective to iteratively expand the residual graph by re-inserting the vertices of some independent set extracted in the first phase and run again the coloring method on the larger residual graph. This method, however, was not tested on graphs larger than a few thousand vertices.

2.3 Peeling-based Approaches

The so-called "peeling" procedure is an efficient scale reduction technique introduced by Abello *et al.* [2] for the maximum clique problem. Since vertices of $(k+1)$-cliques have each at least k neighbors, one can ignore vertices of degree $k-1$ or less. As observed in [27], this procedure corresponds to restricting search to the maximum $(\chi^{low}-1)$-*core* of G where χ^{low} is some lower bound on $\omega(G)$:

Definition 1 (k-Core and denegeracy). *A subset $S \subseteq V$ is called a k-core of the the graph $G = (V, E)$ if the minimum degree of any vertex in the subgraph of G induced by S is k. The minimum value of k for which G has a non-empty k-core is called the* degeneracy *of G.*

As observed by Verma *et al.* [27], the peeling technique can also be used for graph coloring, since low-degree vertices can be colored greedily.

Theorem 1 (Verma *et al.* 2015). *G is k-colorable if and only if the maximum $(k-1)$-core of G is k-colorable.*

Indeed, starting from a k-coloring of the maximum $(k-1)$-core of G, one can explore the vertices of G that do not belong to the core and add them back in an order (the inverse of the *degeneracy ordering*) such that any vertex is preceded by at most $k-1$ of its neighbors. It follows that these extra vertices can each be colored without introducing a $k+1$th color.

This preprocessing technique can be extremely effective on very sparse graphs, and computing a lower bound of the chromatic number is relatively easy: computing the clique number of a graph is NP-hard, but in practice it is much easier than computing its chromatic number. However, the $(\chi^{low}-1)$-core might be too large, and therefore a second use of the peeling technique was proposed in [27]. The idea is to find a coloring of the maximum $(\chi^{up}-2)$-core of G where χ^{up} is an *upper bound* on $\chi(G)$. The $(\chi^{up}-2)$-core has several good properties: it is often small, its chromatic number is a lower bound on $\chi(G)$, and if there exists such a k-coloring with $k < \chi^{up}$, then it can be extended, in the worst case, to a $(\chi^{up}-1)$-coloring of G.

Therefore, Verma *et al.* proposed the following method: Starting from the bounds $\chi^{low} \leq \chi(G) \leq \chi^{up}$, the algorithm solves the maximum $(\chi^{up}-2)$-core of

G to optimality, and extends the corresponding k-coloring greedily following the inverse degeneracy order to a k'-coloring. Then it sets χ^{low} to $\max(\chi^{low}, k)$ and χ^{up} to k'. The algorithm converges since since χ^{low} cannot decrease and χ^{up} is guaranteed to decrease at each step.

Unfortunately, some graphs simply do not have small k-cores, even for k larger than their chromatic number, so this method is limited to extremely sparse graphs. Moreover, notice that the core must be solved to optimality in order to extract relevant information from the iteration and converge.

The approach of Lin *et al.* [16] also uses peeling, but in a slightly different way. A *degree-bounded independent set* is an independent set whose vertices all have a degree strictly smaller than a lower bound χ^{low}on the chromatic number. Their method iteratively finds a maximal clique using a very effective sampling-based heuristic; removes a k-bounded independent set where k is the size of the clique from the graph; and runs DSATUR in order to find an upper bound.

This method is very effective, outperforming the approach of Verma *et al.* on graphs with large cores. However, notice that the vertices in a k-bounded independent set cannot be in a $(k-1)$-core, and therefore this variant of peeling is less effective than Verma's. The two main components are the method to find a clique and the DSATUR heuristic to find upper bounds. The former essentially samples a set of vertices to be expanded to a maximal clique. When extending a clique, a number p of neigbors are probed and the one that maximizes the size of the residual candidate set of vertices to expand the clique is chosen. Several runs are performed with the parameter p growing exponentially at every run. The latter is randomized and augmented with the *recolor* technique [23]: when a new color class i is created for a vertex v, if there exist two color classes C_j, C_k with $j < k$ and a vertex u such that $N(v) \cap C_j = \{u\}$ and $N(u) \cap C_k = \emptyset$, then v and u can be recolored to j and k respectively, thus leaving the color i free.

3 Iterated Dsatur

The overwhelmingly most common lower bound technique is to find a large clique. Several other lower bounds have been used. For instance, two extra lower bounds were proposed in [9]: the Lovász Theta number [17] and a second lower bound based on a mapping between coloring and independent sets on a reformulation of the graph [7]. Another lower bound based on finding embedded Mycielskian graphs [21] was proposed in [11]. Moroever, the bounds obtained by linear relaxation of either the standard model or the set covering problem from the branch & price approach are very strong. However, it is difficult to make any of these methods scale up to graphs with millions of vertices.

Many graphs of the `dimacs10` and `snap` datasets have a chromatic number equal to their clique number. Morever, finding a maximum clique turns out to be much easier in practice than solving the VCP. Therefore, it is often possible to find a maximum clique and they often provide a good lower bound.

In this section, we introduce a method to solve the VCP that scales up to very large graphs. Moreover, it may compute non-trivial lower bounds, that is, larger

Algorithm 1: Iterated Dsatur

Algorithm: `I-Dsatur`

Data: Graph G, Initial order O^0, color assignment C^0 and bounds χ^{low}, χ^{up}

Result: $\chi(G)$

$i \leftarrow 0$

while $\chi^{low} < \chi^{up}$ **do**

  $p \leftarrow 1 + \max\{j \mid C^i(o^i_k) \leq \chi^{low}\ \forall k < j\}$

  $O^{i+1} \leftarrow \{o^i_1, \dots o^i_p\}$

  $i \leftarrow i + 1$

  $C_{core} = \texttt{ExactColoring}(G_{O^i})$

  if $\max(C_{core}) > \chi^{low}$ **then**

    $\chi^{low} \leftarrow \max(C_{core})$

    $C^i \leftarrow C^{i-1}$

  else

    $C^i \leftarrow C_{core}$

    $(O^i, C^i) \leftarrow \texttt{Dsatur}(O^i, C^i)$

    if $\max(C^i) < \chi^{up}$ **then**

      $\chi^{up} \leftarrow \max(C')$

return (χ^{low}) `//` $= \chi(G)$

than the clique number. As a consequence, this method can sometimes produce optimality proofs for extremely large graphs, even when $\omega(G) < \chi(G)$. The principle is to iteratively compute a coloring with DSATUR, and optimize its prefix up to the first occurrence of the color $\chi^{low} + 1$. If there exists a coloring of that subgraph, then the next iteration of DSATUR will follow the optimized prefix, whose length will thus increase. Otherwise, the lower bound can be incremented.

Algorithm 1 uses a variant of DSATUR which can take into account a given total order O of a subset of the vertices and a coloring C for these vertices (it may be a coloring of more vertices, but it is ignored for vertices not in O). This variant assigns first vertices in the given order using the given coloring. Once those vertices have been colored, it proceeds to color the rest of the vertices of the graph using the standard DSATUR heuristic. The modified `Dsatur` returns not only the coloring but also the total order of V that it used to produce this coloring. In the following, we use $O = \langle o_1, \dots, o_p \rangle$ to denote an order of a subset of the vertices, where $\{o_1, \dots, o_p\} \subseteq V$. Moreover, given a coloring C, we write $\max(C)$ for the maximum color used, and $C(v)$ for the color assigned to v.

Algorithm 1 proceeds as follows. Given initial bounds χ^{low} and χ^{up}, as well as a coloring and ordering that witness the upper bound, we extract the *core graph*, which is the subgraph G_{O^1} of G induced by the vertices $\{o_1, \dots, o_p\}$ where p is the maximum index for which all vertices $o_1, \dots, o_{p-1}$ are assigned colors in $[1, \chi^{low}]$. In other words, p is the index of the first vertex that is assigned a

color greater than the current lower bound χ^{low}. The order of these p vertices is fixed for all subsequent runs of DSATUR. We then compute $\chi(G_{O^1})$, using any exact coloring algorithm. In our implementation this is the satisfiability-based algorithm from [11]. If $\chi(G_{O^1}) > \chi^{low}$ then we can update $\chi^{low} = \chi(G_{O^1})$. This is because G_{O^1} is an induced subgraph of G, so $\chi(G_{O^1})$ is a lower bound on $\chi(G)$. On the other hand, if $\chi(G_{O^1}) \leq \chi^{low}$, we fix the first p vertices to their order and color them as in the optimal coloring of G_{O^1} and use them as the starting point for a run of `Dsatur`. In either case, we proceed to the next iteration.

Algorithm 1 converges because at every iteration a growing subset of the vertices have their order fixed and included in the core. Indeed, if $\chi(G_{O^i}) > \chi^{low}$, then the lower bound is increased, which means that more vertices with be in $G_{O^{i+1}}$. On the other hand, if $\chi(G_{O^i}) \leq \chi^{low}$, then the next run of `Dsatur` will be constrained to assign at least o_p to a color in $[1, \chi^{low}]$, so the core graph at the next iteration contains at least one more vertex. In the extreme, the algorithm will terminate when $G_{O^i} = G$.

4 Local Search for Massive Graphs

As far as we know, the best upper bound for the datasets we consider were obtained using either Brelaz' heuristic [16], or by greedily extending the optimal solution of a k-core [27]. Therefore, whether, and to which extent, local search can help in such a case remains to be seen. In this section we describe the modifications we made to Blöchliger and Zufferey's tabu-search algorithm in order to adapt it to extremely large graphs.

Initialization A first very modest, but significant, addition is a method to efficiently initialize the solution of the local search. The algorithm described in [4] is given an integer k and tries to find a k-coloring. Since our method produces colorings during preprocessing (from the computation of the degeneracy ordering and from DSATUR) it is immediate to initialize the solution with such a coloring whereby the vertices of any one color class are considered "uncolored". However, we observed that it was important to choose a small color class, as they can be extremely unbalanced and chosing randomly could lead to a prohibitively large neighborhood to explore in the initial steps.

Chained Flat Moves Recall that a move consists in swapping a node v from the set U of uncolored vertices with its neighbors $N(v) \cap C_i$ in some color class i. When $N(v) \cap C_i = \emptyset$ this is an *improving move* as we have one less uncolored node. Now let call a move (v, i) such that $|N(v) \cap C_i| = \{u\}$ a *flat move*. We know that no strictly improving move was possible, so if there is an improving or a flat move involving u it is likely to be selected next. Therefore, in the event of a flat move we greedily follow chains of flat moves from the previous vertex until reaching an improving move, or until no flat or improving move is possible for that vertex. This technique does not change the neighborhood, but allows to explore it in a more greedy way and is often beneficial. Moreover, we observed

that it was relatively easy to assess if such moves were effective, by counting how many of them lead to an improving move, and by checking their length.

Algorithm 2: Local Search

```
Algorithm: TabuSearch
Data: Graph G = (V, E), Coloring C, Parameters I, t
Result: A coloring of G
best ← C, k ← 0
foreach v ∈ V', 1 ≤ i ≤ max(C) do T_v^i = 0
while k ≤ I do
1   c ← arg min_i(|C_i|)
    U ← C_c
    while i ≤ I and C_i ≠ ∅ do
        v, i ← arg min_{u∈U, j≠c | T_u^j ≤ k}(|N(u) ∩ C_j|)
2       if |N(v) ∩ C_i| = 1 then
            repeat
                C(v) ← i
                v' ← v, i' ← i
                v, i ← arg min_{u∈C_i', j∉{c,i'} | T_u^j ≤ k}(|N(u) ∩ C_j|)
            until |N(v) ∩ C_i| = 1
            if |N(v) ∩ C_i| > 1 then
                C(v) ← c
                T_v^i' ← k + t
3       else
            C(v) ← i
            foreach u ∈ N(v) ∩ C_i do
                C(u) ← c
                T_u^i ← k + t
        k ← k + 1
    if U = ∅ then best ← C
    return best
```

Algorithm 2 is a pseudo-code of our implementation of Blöchliger and Zufferey's tabu search. We denote C_i the set of vertices of color i, that is $C_i = \{v \mid C(v) = i\}$. The outer loop and the color selection in line 1 are not in the original implementation, as well as the random path of flat moves corresponding to the lines between 2 and 3. Notice that ties are broken randomly in every "arg min" operator. Moreover, the management of the tabu list (T_v^i) as well as of the iteration limit, and the choice of applying a random path move is more complex than the pseudo-code shows. We set the parameters as follows.

Tabu list Here we used a relatively straightforward scheme which is in fact a simplified version of what is done in the original code. Every 10000 iterations, the tabu tenure parameter t is decremented, unless it is null or the delta between the lowest and largest size for U (the set of "uncolored" vertices) is lower than or equal to 1 since the last update of the tabu tenure. In both of the latter cases, t is increased by its initial value (the initial value was 10 in all our experiments).

Iteration limit In order to dynamically adapt the number of iterations to the progress made by the tabu search, we used the following policy: Let k be the current number of iterations and I the current limit. When the limit is reached within the outer loop, we check if there was any progress on the upper bound χ^{up} since the last limit update. If there was some progress, then we increase the limit by the current number of iterations ($I = I + k$). Now, let δ be the value of $I - k$ at the start of the inner loop. When the limit is reached within the inner loop, we check if there was any progress on the number of uncolored vertices ($|U|$) since the last limit update. If there was some progress, then we increase the limit by δ, otherwise we increase it by $\delta/2$. We used an initial limit of 250000.

Limit on chains of flat moves In some cases it is possible to explore very long paths of flat moves hence slowing down the algorithm. We introduce a parameter p (originally set to 1) controling the probablity $1/p$ of prefering such moves. Then we simply check the average length l of these moves and their frequency f and adjust p in consequence. In practice, we double p when $l \times f \geq 20$ and decrement it when it is strictly greater than 1 and $l \times f \leq 3$.

5 Overall Approach

Our approach combines the peeling preprocessing from section 2, the tabu search described in section 4 and the iterated DSATUR scheme described in section 3.

The principle we use for choosing the exact sequence of techniques is to apply first those that have the greatest effect for the least computational cost. Therefore, we first compute an upper bound by computing the degeneracy and a lower bound by finding a clique. Although finding the maximum clique is NP-hard, it turns out to be much easier than coloring in the dataset we used, so we solve the problem exactly rather than use a heuristic. It also has a great effect on the rest of the algorithm, as a better initial lower bound results in greater scale reduction using peeling and hence improves all heuristics used further on. After peeling, we first improve the upper bounds using DSATUR and then our local search algorithm. Finally, we switch to iterated DSATUR (`I-Dsatur`), which is exact and hence the most computationally expensive part of the algorithm.

One complication is that the iterated DSATUR phase is initialized with the current best solution. If this solution was found by the local search algorithm, there is no ordering that `I-Dsatur` can use to extract a core. We can produce a relevant ordering from the local search solution simply by sorting the vertices by saturation degree *within the local search coloring*[4] as shown in line 2. However,

[4] ties broken by overall degree

Algorithm 3: Graph Coloring

Algorithm: `LS+I-Dsatur`

Data: Graph $G = (V, E)$, Parameters I, t
Result: The chromatic number of G

`/* Preprocessing phase */`
1 $(O, D) \leftarrow$ `DegeneracyOrder`(G)
$\chi^{up} \leftarrow \max(D) + 1$
$\chi^{low} \leftarrow |$`FindClique`$(G)|$
$H \leftarrow$ subgraph of G induced by $\{o_k, \ldots, o_{|V|}\}$ with
$k = \max\{i \mid j \geq i \text{ or } D(j) < \chi^{low}\}$
$(O, C) \leftarrow$ `Dsatur`(H)
$\chi^{up} \leftarrow \max(\chi^{up}, \max(C))$

`/* Local search phase */`
$C \leftarrow$ `TabuSearch`$(H,\ C,\ I,\ t)$
$\chi^{up} \leftarrow \min(\chi^{up}, \max(C))$
foreach $v \in V'$ **do** $\delta^{sat}(v) \leftarrow |\{C(u) \mid u \in N(v)\}|$
2 $O = \{o_1, \ldots, o_{|V'|}\}$ with $i < j \implies \delta^{sat}(o_i) \geq \delta^{sat}(o_j)$

`/* Iterated` DSATUR `phase */`
$(O, C') \leftarrow$ `Dsatur`$(H,\ O,\ C)$
return `I-Dsatur`$(H,\ O,\ C',\ \chi^{low},\ \chi^{up})$

this coloring may not use the smallest colors for the first vertices in the order, therefore, we apply the following transformation:

We run `Dsatur` following the ordering O. When processing node v, we check if the color $C(v)$ assigned by the tabu search to v has already been mapped to some color, if not, we map it to the minimum color c that v can take and assign c to v. We do the same if the color $C(v)$ happens to be already mapped to c. Otherwise, we switch to the standard DSATUR from that point on.

The resulting coloring is similar (at least in the prefix) to the LS solution, however it is in a form that might have been produced by DSATUR.

6 Experimental Results

Our implementaton uses `dOmega` [28] for finding the initial maximum clique, and `MiniCSP`[5] as the underlying CDCL CSP solver during the I-DSATUR phase.[6]

We compare it to the state of the art: the `FastColor` approach [16]. Unfortunately, we could not compare with the approach described in [27] since the

[5] Sources available at: `https://bitbucket.org/gkatsi/minicsp`.
[6] Sources available at: `https://bitbucket.org/gkatsi/gc-cdcl/src/master/`.

coloring part of this code is now lost.[7] However, this latter approach is dominated by `FastColor` on instances with large cores, hence the hardest.

Every method was run 20 times with different random seeds and with a time limit of one hour and a memory limit of 10GB. The memory limit was an issue only for `dOmega` which exceeded the memory limit on 3 instances. We raised the limit to 50GB in these three cases. We used 4 cluster nodes, each with 35 Intel Xeon CPU E5-2695 v4 2.10GHz cores running Linux Ubuntu 16.04.4.

			`FastColor`			`LS+I-Dsatur`		
	$\|V\|/\|E\|$	(scaled)	CPU time (ms)			CPU time (ms)		
			min	avg	max	min	avg	max
as-22july06	23K/48K	168/3115	**13**	**17**	**21**	3182	4356	6354
caidaRouterLevel	192K/609K	3425/65K	**265**	**375**	**662**	441	7582	40801
citationCiteseer	268K/1157K	6731/76K	533	933	1895	**382**	**516**	**607**
cnr-2000	326K/2739K	86/3652	1952	2250	2526	**362**	**398**	**448**
coAuthorsCiteseer	227K/814K	87/3741	**95**	**155**	**243**	223	289	356
coAuthorsDBLP	299K/978K	115/6555	**161**	**254**	**410**	356	425	518
coPapersCiteseer	100K/498K	130/4160	**32**	**46**	**83**	68	90	109
coPapersDBLP	540K/15M	337/57K	**1074**	**1238**	**1419**	1667	2037	2564
cond-mat-2005	40K/176K	30/435	**12**	**32**	**51**	28	39	55
eu-2005	333K/3949K	2414/123K	3396	3797	4617	**557**	**651**	**709**
in-2004	163K/2602K	343/59K	731	1664	1995	**228**	**262**	**310**
rgg-n-2-17-s0	131K/729K	34/262	**107**	210	376	159	**202**	**242**
rgg-n-2-19-s0	524K/3270K	19/170	**582**	1365	1768	849	**1107**	**1357**
rgg-n-2-20-s0	1049K/6892K	172/1624	**1335**	**2607**	3917	2042	2659	**3443**
rgg-n-2-21-s0	2097K/14M	19/171	5442	9373	13240	**4975**	**6311**	**8043**
rgg-n-2-22-s0	4194K/30M	20/190	**8119**	21840	27280	11900	**13819**	**17512**
rgg-n-2-23-s0	8389K/64M	22/230	**18465**	51959	67686	25125	**31100**	**37256**
rgg-n-2-24-s0	17M/133M	82/994	**39562**	94233	136666	49909	**61154**	**80953**
belgium_osm	1441K/1550K	1239K/1348K	**239**	**270**	**326**	1039	1240	1527
ecology1	1000K/1998K	1000K/1998K	**483**	**536**	**943**	1428	1561	1702
luxembourg_osm	115K/120K	93K/98K	**9**	**16**	**41**	47	65	81
preferentialAttachment	100K/500K	100K/500K	182	978	2574	**137**	**175**	**243**
Average CPU time			**3763**	8825	12215	4777	**6184**	**9327**

Table 1: CPU Time (easy `dimacs10` instances)

The first two columns of tables 1, 2, 3 and 5 give the size of the graph (number of vertices/edges) before and after scale reduction. In all these tables, bold font is used to highlight the (strictly) best outcomes. In tables 1 and 2 we report the CPU time in milliseconds for the "easy" instances of the `dimacs10` and `snap` sets, respectively. We say that an instance is easy when both `I-Dsatur` and `FastColor` solved to optimality. We give the minimum, maximum and average CPU time – parsing excluded – across the 20 random runs on the same instance.

Tables 3 and 5 show the lower (χ^{low}) and upper bounds (χ^{up}) found by `I-Dsatur` and `FastColor` on the rest of the dataset ("hard" instances). Both for the lower and upper bound, we give the best and average value across the 20 random runs on the same instance. We use an asterisk (*) to denote that

[7] Personnal communication with the authors.

	$\|V\|/\|E\|$	(scaled)	FastColor			LS+I-Dsatur		
			CPU time (ms)			CPU time (ms)		
			min	avg	max	min	avg	max
as-skitter	1696K/11M	4567/328K	**10151**	**11600**	**13087**	25810	39951	75655
ca-AstroPh	19K/198K	57/1596	**13**	30	56	21	**27**	**36**
ca-CondMat	23K/93K	26/325	9	**16**	**21**	**8**	19	27
ca-GrQc	5246/14K	44/946	1	3	11	**0**	**2**	**3**
ca-HepPh	12K/118K	239/28K	39	44	78	**6**	**13**	**16**
ca-HepTh	9880/26K	32/496	**1**	**4**	6	2	5	6
athletes_edges	14K/87K	42/793	**7**	**13**	**20**	11	17	22
com-amazon.ungraph	335K/926K	497/1683	**152**	**248**	**486**	413	529	625
com-dblp.ungraph	317K/1050K	114/6441	**130**	**245**	**453**	358	453	588
com-lj.ungraph	3925K/34M	383/73K	23748	45137	62572	**15545**	**22862**	**28152**
company_edges	14K/52K	65/842	4	**9**	20	4	9	**15**
government_edges	7057/89K	1008/30K	**5**	**24**	**53**	39	63	81
new_sites_edges	28K/206K	36/615	**17**	41	115	32	**38**	**46**
politician_edges	5908/42K	545/11K	**26**	**47**	**74**	4546	5218	5961
public_figure_edges	12K/67K	586/17K	**12**	**48**	75	40	52	**64**
tvshow_edges	3892/17K	61/1820	1	2	6	1	**2**	**3**
wiki-topcats	1788K/25M	114K/5459K	**20797**	**42787**	**66729**	84592	100965	118302
loc-gowalla_edges	197K/950K	3768/130K	**206**	**492**	1007	459	558	**652**
loc-gowalla_totalCheckins	5669K/6442K	5669K/6442K	**4895**	**5982**	**7736**	6793	8195	10213
Amazon0302	262K/900K	286/950	**154**	**301**	**444**	269	445	598
Amazon0312	401K/2350K	27K/179K	**409**	**556**	**771**	802	1011	1348
Amazon0505	410K/2439K	33K/219K	**495**	**586**	**725**	815	999	1237
Amazon0601	403K/2443K	33K/221K	**445**	**580**	**922**	816	1053	1392
roadNet-CA	1965K/2767K	4568/7572	**459**	**721**	**1314**	2020	2314	3054
roadNet-PA	1088K/1542K	952/1547	**258**	**473**	**1263**	1079	1279	1629
roadNet-TX	1380K/1922K	1579/2637	**308**	**451**	**1198**	1378	1634	2103
soc-sign-epinions	132K/711K	253/22K	365	1068	1605	**280**	**332**	**391**
HU_edges	48K/223K	72/612	**25**	110	183	50	**62**	**83**
RO_edges	42K/126K	1638/7982	**16**	44	68	32	**39**	**51**
soc-LiveJournal1	4847K/43M	474/106K	60025	98270	113546	**23413**	**29647**	**38846**
soc-pokec-relationships	1633K/22M	288K/9034K	**9041**	43465	116745	20340	**26412**	**32036**
twitter_combined	81K/1342K	719/49K	1181	1653	2192	**337**	**384**	**415**
web-BerkStan	685K/6649K	392/41K	4849	5103	6633	**906**	**1068**	**1291**
web-Google	876K/4322K	103/2513	**860**	**1373**	**1961**	1287	1752	2259
web-NotreDame	326K/1090K	1367/108K	**126**	**154**	**241**	316	354	434
web-Stanford	282K/1993K	1370/79K	885	1317	1526	**505**	**578**	**667**
wiki-RfA	38K/94K	38K/94K	**47**	**51**	**53**	75	102	124
Average CPU time			**3788**	7109	10919	5227	**6715**	**8876**

Table 2: CPU Time (easy snap instances)

the maximum lower bound found over the 20 runs is as high as the minimum upper bound, signifying that the method is able close the instance. Moreover, for the results of I-Dsatur, we denote via a superscript in which phase of the approach the best outcome was found. A value of 0 stands for the computation of the degeneracy ordering, 1 for the preprocessing phase, 2 for the local search and 3 for the iterated DSATUR phase.

Finally, tables 4 and 6 give a summary view for hard instances, of respectively the dimacs10 and snap datasets, with the arithmetic and geometric mean bounds; overall ratio of optimality; and overall mean CPU time.

We first observe that for many of these graphs (see tables 1 and 2) finding an optimal coloring is easy. One reason is that their clique and chromatic num-

	$\|V\|/\|E\|$	(scaled)	FastColor				LS+I-Dsatur			
			χ^{low}		χ^{up}		χ^{low}		χ^{up}	
			max	avg	min	avg	max	avg	min	avg
kron_g500-logn16	55K/2456K	6885/1495K	136	136.00	151	**152.45**	^{1}136	136.00	^{3}**145**	153.40
333SP	3713K/11M	3713K/11M	4	4.00	5	5.00	^{1}4	4.00	^{0}5	5.00
G_n_pin_pout	100K/501K	100K/501K	4	4.00	6	6.00	^{3}4	4.00	^{2}**5**	**5.00**
audikw1	944K/38M	938K/38M	36	36.00	40	40.85	^{1}36	36.00	^{2}**39**	**39.05**
cage15	5155K/47M	5135K/47M	6	6.00	12	12.00	^{1}6	6.00	^{2}**11**	**11.05**
ldoor	952K/23M	952K/23M	21	21.00	32	32.70	^{3}**23**	**21.65**	^{2}**28**	**29.85**
smallworld	100K/500K	100K/500K	6	6.00	7	7.00	^{1*}6	6.00	^{2}**6**	**6.00**
wave	156K/1059K	156K/1059K	6	6.00	8	8.00	^{1}6	6.00	^{1}8	8.00

Table 3: Lower and Upper Bounds (hard dimacs10 instances)

method	χ^{low}		χ^{up}		Opt.	CPU
	avg	avg (G)	avg	avg (G)	avg	avg
LS+I-Dsatur	27.456	11.762	32.169	14.853	0.125	658034
FastColor	27.375	11.717	33.000	15.910	0.000	324928

Table 4: Summary (hard dimacs10 instances)

bers are equal. However, this is also the case for some graphs classified here as "hard". Whereas we use a complete maximum clique algorithm in our approach, FastColor does not and yet it finds a maximum clique in all the "easy" graphs and in most of the "hard" ones. Moreover, both solvers were able to quickly find a maximum clique and an optimal coloring. In particular, we can see that many easy graphs are solved during the preprocessing phase, the maximum $(\chi^{low}-1)$-core being very small. Those graphs are therefore trivial both for FastColor and for our approach, which are in fact similar on those. There is a slight advantage to our method in terms of average run time, both for easy dimacs10 and easy snap instances, which can presumably be attributed to our peeling method being more efficient than the independent set extraction in FastColor.

Of the hard dimacs10 instances in table 3, all but kron_g500-logn16 are quasi-regular, i.e., every vertex has roughly the same degree. These graphs do not have small cores, hence the peeling phase is irrelevant. We can see that on these graphs, the tabu search algorithm significantly outperforms DSATUR and therefore our approach dominates FastColor for the upper bound. For instance, on ldoor, LS+I-Dsatur finds a 29.85-coloring on average whereas the best coloring found by FastColor has 32 colors. On the instance kron_g500-logn16, the tabu search performs poorly and is on average dominated by FastColor. In one run, however, the iterated DSATUR algorithm is able to find a much better coloring using 6 fewer colors than the best one found by FastColor. The aggre-

	$\|V\|/\|E\|$	(scaled)	FastColor χ^{low} max	FastColor χ^{low} avg	FastColor χ^{up} min	FastColor χ^{up} avg	LS+I-Dsatur χ^{low} max	LS+I-Dsatur χ^{low} avg	LS+I-Dsatur χ^{up} min	LS+I-Dsatur χ^{up} avg
cit-HepPh	35K/421K	10K/215K	*19	19.00	19	**19.00**	[1]*19	19.00	[3]19	19.10
cit-HepTh	28K/352K	7278/198K	*23	**23.00**	**23**	**23.65**	[3]23	22.15	[3]24	24.05
artist_edges	51K/819K	19K/606K	18	18.00	**19**	**19.95**	[1]18	18.00	[3]20	20.00
com-orkut.ungraph	3072K/117M	784K/59M	50	49.45	74	77.25	[1]**51**	**51.00**	[3]**71**	**72.90**
com-youtube.ungraph	1135K/2988K	30K/749K	17	17.00	23	**23.00**	[3]**19**	**17.95**	[3]23	23.25
email-Eu-core	986/16K	552/13K	18	18.00	19	**19.00**	[3]*__19__	**19.00**	[3]19	19.15
email-Enron	37K/184K	2873/79K	20	**20.00**	**23**	**23.45**	[3]20	19.30	[1]24	24.00
email-EuAll	265K/364K	1691/42K	16	16.00	18	**18.00**	[3]*__18__	**17.70**	[3]18	18.40
p2p-Gnutella04	11K/40K	8379/37K	4	4.00	5	5.00	[3]4	4.00	[1]5	5.00
p2p-Gnutella05	8850/32K	5755/28K	4	4.00	5	5.00	[1]4	4.00	[1]5	5.00
p2p-Gnutella06	8717/32K	6727/30K	4	4.00	5	5.00	[3]4	4.00	[1]5	5.00
p2p-Gnutella08	6301/21K	3051/15K	5	5.00	6	6.00	[3]*__6__	**6.00**	[2]6	6.00
p2p-Gnutella09	8114/26K	4448/21K	5	5.00	6	6.00	[3]*__6__	**6.00**	[1]6	6.00
p2p-Gnutella24	27K/65K	16K/54K	4	**4.00**	5	5.00	[3]4	3.70	[1]5	5.00
p2p-Gnutella25	23K/55K	9764/38K	4	4.00	5	5.00	[1]4	4.00	[1]5	5.00
p2p-Gnutella30	37K/88K	15K/61K	4	4.00	5	5.00	[1]4	4.00	[1]5	5.00
p2p-Gnutella31	63K/148K	24K/100K	4	4.00	5	5.00	[1]4	4.00	[1]5	5.00
soc-sign-Slashdot081106	77K/469K	5042/171K	26	26.00	29	**29.00**	[3]*__29__	**28.90**	[3]29	29.05
soc-sign-Slashdot090216	82K/498K	4960/171K	27	27.00	29	**29.00**	[3]*__29__	**29.00**	[3]29	29.05
soc-sign-Slashdot090221	82K/500K	4984/173K	27	27.00	29	**29.00**	[3]*__29__	**28.90**	[3]29	29.20
soc-sign-bitcoinalpha	3783/14K	472/5991	10	10.00	12	12.00	[3]*__12__	**12.00**	[1]12	12.00
soc-sign-bitcoinotc	5881/21K	564/8017	11	11.00	12	12.00	[3]*__12__	**12.00**	[3]12	12.00
HR_edges	55K/498K	23K/329K	12	12.00	13	13.00	[1]12	12.00	[2]13	13.00
Wiki-Vote	7115/101K	2316/84K	17	17.00	22	**22.00**	[3]**19**	**18.70**	[3]22	22.20
facebook_combined	4039/88K	487/30K	69	69.00	70	**70.00**	[3]*__70__	**70.00**	[3]70	70.10
gplus_combined	108K/12M	13K/6838K	325	324.10	327	327.75	[3]*__326__	**324.60**	[3]**326**	**327.65**
soc-Epinions1	76K/406K	5004/210K	23	23.00	**28**	**28.00**	[3]**26**	**24.25**	[1]29	29.00
CollegeMsg	1899/14K	1011/12K	7	7.00	9	9.00	[3]*__9__	**8.40**	[1]9	9.00
sx-askubuntu	157K/456K	1964/62K	23	23.00	25	25.00	[3]*__24__	**24.00**	[3]**24**	**24.95**
sx-mathoverflow	25K/188K	1638/81K	30	30.00	**35**	**35.95**	[3]**32**	**31.95**	[3]36	36.20
sx-stackoverflow	2584K/28M	114K/11M	55	55.00	**66**	**66.15**	[1]55	55.00	[1]67	67.00
sx-superuser	192K/715K	3041/123K	29	29.00	30	30.00	[3]*__30__	**30.00**	[3]30	30.00
wiki-talk-temporal	1094K/2788K	12K/656K	25	25.00	46	**46.00**	[3]**26**	**25.20**	[3]46	46.20
wiki-Talk	2394K/4660K	16K/794K	26	26.00	48	**48.25**	[3]**29**	**28.50**	[3]48	48.90
wiki-Vote	7120/101K	2316/84K	17	17.00	22	22.00	[3]**19**	**18.95**	[3]22	22.00

Table 5: Lower and Upper Bounds (hard snap instances)

method	χ^{low} avg	χ^{low} avg (G)	χ^{up} avg	χ^{up} avg (G)	Opt. avg	CPU avg
LS+I-Dsatur	28.747	16.036	32.153	18.452	0.334	279715
FastColor	27.901	15.299	32.126	18.388	0.039	194104

Table 6: Summary (hard snap instances)

gated results given in table 4 show that LS+I-Dsatur outperforms FastColor both for the lower and upper bounds on this dataset.

The iterated DSATUR algorithm is also able to improve the lower bound of 2 instances out of 8 (`ldoor` and `G_n_pin_pout`). However, for the latter, `FastColor` produces the same lower bound (4) which is larger than the maximum clique found by `dOmega`. We do not know how to explain this.

On hard instances of the `snap` dataset (table 5), the picture is very different with in particular the tabu search being almost useless. The best coloring found by our method was obtained during the local search phase only once, for the instance `HR_edges`. In all other cases the best coloring was produced either during preprocessing via DSATUR, or during the iterated DSATUR phase. Overall, as shown in table 6, this is slightly less efficient for the upper bound than `FastColor` which repreatedly uses DSATUR and eventually finds better colorings in several instances whilst `LS+I-Dsatur` is best only on four instances.

The iterated DSATUR phase, however, is very effective with respect to the lower bound. It improves on the maximum clique found by `dOmega` in 26 out of 35 instances, and it matches the best upper bound for 14 instances. Here again, we observe on three instances (`cit-HepTh`, `email-Enron` and `p2p-Gnutella24`) that `FastColor` outputs a lower bound greater than that found by `dOmega`. Overall, our approach can close 15 of the hard instances whereas `FastColor` can only close two of them. For 11 of these instances[8], the optimal coloring was not known prior to this study, as far as we know.

7 Conclusions

We have presented a new algorithm for exactly computing the chromatic number of large real world graphs. This scheme combines a novel local search component that performs well on massive graphs and gives improved upper bounds as well as an iterative reduction method that produces much smaller graphs than previous state of the art scale reduction methods. This scheme involves extracting more information than simply a coloring from the DSATUR greedy coloring heuristic and iteratively solving reduced instances with a complete, branch-and-bound solver, in such a way that lower bounds produced for the reduced graphs are also lower bounds of the original graph. Combined with the fact that we achieve more significant reduction than the current state of the art means that we can find non-trivial lower bounds even when peeling-based reduction cannot reduce the graph to fewer than hundreds of thousands of vertices. Indeed, in our experimental evaluation on a set of massive graphs, this method is able to produce both better lower and upper bounds than existing solvers and proves optimality on several (almost 75%) of them.

We expect that finding a method to extract cores from other heuristics, such as our local search procedure will further improve performance.

[8] `email-Eu-core`, `email-EuAll`, `Gnutella08/09`, `bitcoinalpha`, `bitcoinotc`, `facebook`, `gplus`, `CollegeMsg`, `sx-askubuntu` and `sx-superuser`

References

1. Karen I. Aardal, Stan P.M. Van Hoesel, Arie M.C.A. Koster, Carlo Mannino, and Antonio Sassano. Models and solution techniques for frequency assignment problems. *Annals of Operations Research*, 153(1):79–129, 2007.
2. James Abello, Panos Pardalos, and Mauricio G. C. Resende. External memory algorithms. In James M. Abello and Jeffrey Scott Vitter, editors, *In External Memory Algorithms*, chapter On Maximum Clique Problems in Very Large Graphs, pages 119–130. American Mathematical Society, Boston, MA, USA, 1999.
3. David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering. http://www.cc.gatech.edu/dimacs10/., 2012.
4. Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Comput. Oper. Res.*, 35(3):960–975, March 2008.
5. Daniel Brélaz. New Methods to Color the Vertices of a Graph. *Commun. ACM*, 22(4):251–256, 1979.
6. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, January 1981.
7. Denis Cornaz and Vincent Jost. A one-to-one correspondence between colorings and stable sets. *Operations Research Letters*, 36(6):673 – 676, 2008.
8. Dominique de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19(2):151 – 162, 1985.
9. Fabio Furini, Virginie Gabrel, and Ian-Christopher Ternier. Lower bounding techniques for dsatur-based branch and bound. *Electronic Notes in Discrete Mathematics*, 52:149 – 156, 2016. INOC 2015 – 7th International Network Optimization Conference.
10. Jin-Kao Hao and Qinghua Wu. Improving the extraction and expansion method for large graph coloring. *Discrete Appl. Math.*, 160(16-17):2397–2407, November 2012.
11. Emmanuel Hebrard and George Katsirelos. Clause Learning and New Bounds for Graph Coloring. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP-2018)*, pages 179–194, 2018.
12. Alain Hertz and Dominique de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, Dec 1987.
13. David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.
14. Frank T. Leighton. A Graph Coloring Algorithm for Large Scheduling Problems. *J. Res. Natl. Bur. Standards*, 84:489–506, 1979.
15. Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.
16. Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. A reduction based method for coloring very large graphs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 517–523, 2017.
17. László Lovász. On the shannon capacity of a graph. *IEEE Trans. Inf. Theor.*, 25(1):1–7, September 2006.
18. Enrico Malaguti, Michele Monaci, and Paolo Toth. An exact approach for the vertex coloring problem. *Discrete Optimization*, 8(2):174 – 190, 2011.

19. Anuj Mehrotra and Michael A. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8:344–354, 1995.
20. Laurent Moalic and Alexandre Gondran. Variations on memetic algorithms for graph coloring problems. *CoRR*, abs/1401.2184, 2014.
21. Jan Mycielski. Sur le coloriage des graphes. In *Colloq. Math*, volume 3, pages 161–162, 1955.
22. Taehoon Park and Chae Y Lee. Application of the graph coloring algorithm to the frequency assignment problem. *Journal of the Operations Research society of Japan*, 39(2):258–265, 1996.
23. Ryan A. Rossi and Nesreen K. Ahmed. Coloring large complex networks. *CoRR*, abs/1403.3448, 2014.
24. Bas Schaafsma, Marijn Heule, and Hans van Maaren. Dynamic Symmetry Breaking by Simulating Zykov Contraction. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT-2009)*, pages 223–236, 2009.
25. Pablo San Segundo. A new dsatur-based algorithm for exact vertex coloring. *Computers & Operations Research*, 39(7):1724 – 1733, 2012.
26. Allen Van Gelder. Another Look at Graph Coloring via Propositional Satisfiability. *Discrete Appl. Math.*, 156(2):230–243, 2008.
27. Anurag Verma, Austin Buchanan, and Sergiy Butenko. Solving the maximum clique and vertex coloring problems on very large sparse networks. *INFORMS J. on Computing*, 27(1):164–177, February 2015.
28. Jose L. Walteros and Austin Buchanan. Why is maximum clique often easy in practice? Optimization Online, `http://www.optimization-online.org/DB_HTML/2018/07/6710.html`, 2018.
29. Zhaoyang Zhou, Chu-Min Li, Chong Huang, and Ruchu Xu. An exact algorithm with learning for the graph coloring problem. *Computers & Operations Research*, 51:282–301, 2014.